Thomas Foulds (tcf9bj)
Amanda Ray (ajr2fu)
03-28-13
CS4414
Assignment 4

We began our implementation by using the lab 3 reference solution provided on the course Collab site. The initial functions implemented in the reference solution needed to be updated so that a **priority field** (0 for highest priority, 9 for lowest priority) could be established for each uthread. This priority field is necessary to enable us to effectively implement the scheduling scheme requested – priority-based, with Round Robin scheduling within the same priority level. We handled this in several places – by setting the calling thread's pri value to 9 in uthread_init(), by allowing the user to specify a desired pri value in uthread_create() (note – the uthread struct was also updated to reflect a pri value). Within uthread_create(), we also added the ability to quicksort all uthreads by their priority value and arrange them from highest priority to lowest priority (this is similar to the way in which we sort uthread_wait_records, discussed in the section of this report allocated to uthread_mutex_lock()). Within yield, if the current thread has a higher priority (lower pri value) than the next thread, we set the current thread to the head thread if it is not already; if the current thread has a lower priority (higher pri value) than the next thread, we set the thread with higher priority to be the current thread. We chose to design the priority queueing this way as the presence of a single integer value associated with each uthread allows us to easily manipulate their order.

Another main functionality that needed implementation was a **timer/thread scheduler**. We want each thread to have 1ms to execute in; when this time slice expires or a yield is called, we need to make the next scheduling decision (priority-based; Round Robin within the same priority level). We include <sys/time.h> and declare a timer, yield_timer, with an interval of 1 ms (as declared by the static int TIME_SLICE_USEC). We chose to implement this at the beginning of our code as timers are process-wide; it's best to set one up before initiating any threads or mutexes.

The implementation of **void uthread_mutex_init(uthread_mutex_t *lockVar)** is fairly straightforward. As the user is responsible for allocating memory for the mutex lockVar, all that needs to be done here is initialize all attributes of a uthread_mutex to null. As we have defined the uthread_mutex in our uthread.h file, we need a locking_thread (the thread that the mutex locks onto – not necessarily the thread at the head of our waiting queue), an int to represent the length of the wait queue (wait_queue_length – this allows us to keep track of when there are or are not additional threads attempting to access the mutex), and two nodes to represent the head and tail of the wait queue. These two nodes (wait_queue_head and wait_queue_tail) are each of the type uthread_wait_record, and each points to a thread as well as the next uthread_wait_record. This allows us to easily access our threads in priority order, as well as to know when there are no more threads

attempting to access the mutex. We chose to define our mutex in this fashion as it merely requires a few pointers to objects that we have already defined, and a length of a queue. By referencing structs that already exist within our program, we avoid unnecessary labor and complication.

The implementation of **void uthread_mutex_lock(uthread_mutex_t *lockVar)** requires several steps – blocking of signals, determination of the number of threads attempting to lock the mutex, determining which thread may access the mutex (if multiple threads are attempting to do so), and resetting the signal mask. In order to accomplish this, we first must create a set of signals equal to all signals, and use sigprocmask() to block them. We then instantiate a uthread_wait_record that will represent the thread attempting to lock the mutex. If this is the only thread attempting to access the mutex, we can assign both wait_queue_head and wait_queue_tail to be equal to this new_wait; we can also set the locking_thread equal to new_wait's associated thread. This is the simplest case for locking; if there are more threads attempting to access the mutex, we must determine which has priority. We do this by adding the newly-created wait_record to a queue of wait_records and performing a quicksort based on the priority values of each referenced thread. This will order our queue from highest priority (0) to lowest priority (10). We then reset our head and tail pointers to ensure that the queue is ready to be manipulated as necessary, reset our signal mask, and use the uthread_yield() function to ensure that the locking_thread is the one currently running. We chose to only create a new wait record when a thread attempts to access the mutex, thereby avoiding unnecessarily allocating space and making cleanup difficult later. We also chose to sort immediately upon adding a new wait_record to the queue, which will ensure that our queue is always sorted in priority order and that our scheduling algorithm will run appropriately.

The implementation of **void uthread_mutex_unlock(uthread_mutex_t *lockVar)** begins much like the implementation of the lock function – we must block all signals to ensure atomicity of transactions. We must then check for several cases to ensure the most appropriate course of action. If the unlocking thread is the only one waiting, we set a newly-allocated uthread_wait_record (called old_record) to be equal to the unlocking thread's wait_record; we also set the head and tail of the wait_queue to NULL, letting us know that the queue is empty and there are no threads remaining which are attempting to access the mutex. We next check to see if the unlocking thread is at the head of our wait_queue; if it is, we can set old_record equal to this head, and set the next thread to be the new head of the wait_queue. Lastly, we check to see if the current thread is not the highest priority one in the queue. We allocate two new wait_records, search_prev_p and prev_p, and set them equal to the head and head->next nodes, respectively. We then search the wait_queue until we find the spot where search_p->thread is equal to lockVar->locking_thread, and then set old_record equal to this thread. We then allow the next highest priority thread, that at the head of our wait_queue, to access the mutex. Lastly, we unblock the signals (the second portion of our signal blocking at the beginning of the transaction). Our design choices here reflect choices made in

uthread_mutex_lock(); we want to be stylistically consistent across methods. The most difficult case to implement was unlocking the mutex when the locking_thread was not the thread at the head of our wait_queue; by declaring two additional nodes and searching through the wait_queue to find our position, we know which thread in our queue is yielding and can then yield to the head (which is the highest priority thread).

The implementation of **void uthread_mutex_destroy(uthread_mutex_t *lockVar)**, like that of the initialization function, was fairly straightforward. As it is the user's responsibility to free the mutex object lockVar, all the destroy function needs to do is clean up all dynamically allocated internal structures. Since the only dynamically allocated internal structures are those uthread_wait_records we create in uthread_mutex_lock(), we can easily run through the wait_queue and free each node there. By creating a temporary uthread_wait_record node, setting it equal to the head, setting the head uthread_wait_record equal to the next record, and freeing the temp, we are able to free all dynamically allocated memory. Few design decisions truly had to be made here, as this is the simplest way to free a queue of allocated structs while defining only one additional temporary node.